Procedia Computer Science

Volume 80, 2016, Pages 450–460

ICCS 2016. The International Conference on Computational Science

ADAMANT: tools to capture, analyze, and manage data movement

Pietro Cicotti[1] and Laura Carrington[1]

San Diego Supercomputer Center,
University of California, San Diego
pcicotti@sdsc.edu, lcarring@sdsc.edu

Abstract

In the converging world of High Performance Computing and Big Data, moving data is becoming a critical aspect of performance and energy efficiency. In this paper we present the Advanced DAta Movement Analysis Toolkit (ADAMANT), a set of tools to capture and analyze data movement within an application, and to aid in understanding performance and energy efficiency in current and future systems. ADAMANT identifies all the data objects allocated by an application and uses instrumentation modules to monitor relevant events (e.g. cache misses). Finally, ADAMANT produces a per-object performance profile.

In this paper we demonstrate the use of ADAMANT in analyzing three applications, BT, BFS, and Velvet, and evaluate the impact of different memory technology. With the information produced by ADAMANT we were able to model and compare different memory configurations and object placement solutions. In BFS we devised a placement which outperforms caching, while in the other two cases we were able to point out which data objects may be problematic for the configurations explored, and would require refactoring to improve performance.

Keywords: profiling, memory system, caches, modeling, computer architecture

1 Introduction

Data movement is a critical aspect of data-intensive computing, by definition, and going forward it will be a critical aspect of any extreme scale computation. There are a number of essential challenges that stem from the technological changes that drive the cost of moving data higher relative to the cost of executing instructions. This cost inversion holds true both for performance and energy, and projections for future systems suggest that it will continue, at least for a few more technology generations.

This process has been defined as a data *red shift*: the data is moving farther away from the computing as the performance gap between communication latency and computing throughput widens [24]. The so called *Memory Wall* is just evidence of this transition, from systems with slow floating point operations (several cycles per operation) and fast load/store operations (one

 Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2016

doi:10.1016/j.procs.2016.05.323

operation per cycle), to fast floating point operations (several operations completed per cycle) and slow load/store operations (hundreds of cycles per operation) [29].

Several architectural features and workload optimizations compensate for the gap between data movement and computing. For example, a large fraction of on-die resources are dedicated to caches and prefetch engines to hide the latency of data access, often at the expense of energy efficiency [7].

Future systems will employ even more complex, heterogeneous, and deeper memory hierarchies. Beginning with heterogeneous systems and accelerators, which in most cases feature fast memory residing on the accelerator package, it became necessary to transfer data between the memories via PCIe lanes. As a result, managing and optimizing data transfers is of primary importance in order to take advantage of accelerators. Programming models like CUDA provide APIs to explicitly control transfers [8], but do not provide support to optimize transfers; research efforts have explored programming models and domain-specific solutions to manage transfers automatically. Other accelerators feature fast user-managed memory pools. For example, the current Xeon Phi (i.e. Knights Corner [14]) has access to the host memory and a pool of GDDR memory, with the latter being smaller but with greater bandwidth than the host memory; similarly, the next generation Xeon Phi (i.e. Knights Landing) will feature on package MCDRAM memory.

Other solutions are designed to reduce the power draw of the processor, like scratchpad memories and software-managed on-chip memories. In contrast to the situation with caches, that automatically store frequently accessed data, the programmer has to explicitly manage transfers to and from the scratchpads. The advantage is that the structure of scratchpad memories is very simple (compared to the structure of caches) and therefore much more efficient. The disadvantage is that scratchpad memories must be explicitly managed, which is a significant burden for the programmer. Several research efforts explored compiler support and other type of analysis to automatically devise optimal management strategies [25, 27].

Finally, to address the capacity needs of many-core processors, emerging non-volatile memory (NVM) technology that offer close to DRAM level of performance and endurance will be employed to increase the capacity of main memory with virtually no static power draw. The adoption of NVM to expand the capacity of main memory will deepen the memory hierarchy further. In addition, locality becomes even more crucial to avoid performance and energy efficiency penalties, and most NVM technologies have asymmetric read write characteristics: writes are slower and require more energy than reads; optimizing data movement for complex asymmetric memory hierarchy will require dealing with locality as well as access pattern characteristics, including distinguishing read and write operations.

As the hierarchy grows deeper, more complex, and heterogeneous, a greater understanding of data movement within large scale applications is required in order to use the memory hierarchy efficiently. New methodologies and tools are needed to capture and analyze data-movement across all the layers of the hardware/software stack, in order to understand data-movement and devise optimal policies, and eventually control and manage data movement accordingly.

In this paper we present the Advanced DAta Movement Analysis Toolkit (ADAMANT). The tools are a product of our research in capturing, analyzing, and understanding data movement in high performance and data intensive computing. Specifically, ADAMANT includes tools to create per data object characterizations of the data movement within an application, to inform algorithm design and tuning, devise optimal data placement, and to manage data movement improving locality and optimizing performance and efficiency.

The remainder of the paper is organized as follows. Section 2 describes related work. Section 3 describes the design of ADAMANT in detail. In Section 4 we present the results of applying

ADAMANT to analyze three applications and model performance. In Section 5 we conclude and discuss future work.

2 Related Work

In order to characterize the performance and the efficiency of an application, profiling and tracing tools typically focused on instruction-based characterizations. Many tools use instrumentation, both of source code and binary, to insert probes that periodically collect performance information. The Tuning and Analysis Utilities (TAU) [26], the PMaC framework [6], the HPCToolkit [1], and several others [22], are examples of libraries and tools that rely on instrumentation to profile the behavior of the application. Code constructs are characterized by their overall behavior, mostly focusing on cache hit misses, lacking the data object specific information that is necessary when optimizing data placement in complex memory hierarchies.

Recently, the HPCToolkit [18] added a functionality to associate high latency instructions with data objects; while this effort acknowledges the importance of data movement in performance profiling, it still addresses the data movement problem with a perspective that is centered on instructions. Similarly, MemAxes visualizes hardware events associated to long latency memory accesses, and shows the instructions and the data structures involved [12].

Other tools concerned with data movement address costly transfer between nodes in Non-Uniform Memory Access NUMA) architectures [15, 21, 12, 19], and detect problematic access patterns in threaded codes, like false sharing and contention.

In ADAMANT, data objects are central to the profile and its view is centered on understanding the dynamics of data movement across the system and throughout the phases of the program.

Virtually, all profiling tools rely on hardware events, such as cache hits and misses. Both Intel's [13] and AMD's instruction sets architecture [2] define sets of hardware performance counters to collect performance data and estimate the power draw. Most of these are architectural features defined as Model Specific Registers (MSR), Precise Event Based Sampling, Running Average Power Limit (RAPL) [13], and Instruction Based Sampling (IPB) [10]. While it is possible to directly access registers via inline assembly, special devices, and file systems (e.g. the Model Specific Register module or the system file system in Linux), the Performance Application Programmer Interface (PAPI) provides an interface that tool developers and programmers can use to conveniently and portably access hardware performance counters [23].

Tools that do not use hardware performance counters rely on different metrics to quantify locality. Reuse distance is such a platform-independent metric: assuming a given granularity (e.g. cache line size), a reference to a certain address has reuse distance d, where d is the number of unique addresses referenced since the last reference to that address [11, 9]. Reuse distance can be used to characterize locality and approximate the behavior caches (hardware or software). However, while reuse distance can be used to approximate cache hit rates, it is expensive to compute and several tools use sampling to limit the overhead [5, 17].

Hardware counters constitute the foundation of most profiling tools. With ADAMANT we adopt a hybrid approach that leverages both hardware performance counters and simulation. ADAMANT can collect information from different modules based on instrumentation and simulation, as well as hardware counters. The advantage of this approach is flexibility; with simulators it is possible to observe the behavior and collect information otherwise not available in the hardware, and more importantly, it makes it possible to study hardware that is not available or simply does not exist (e.g. prototype evaluation and co-design). The drawback of simulation is the resulting slowdown in the application. For this reason, tools defined to

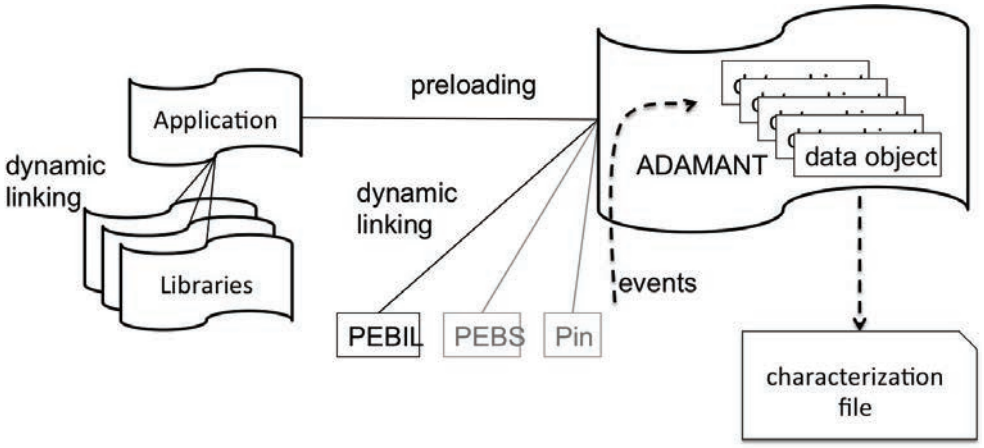


Figure 1: Deployment diagram of ADAMANT's main library and modules.

use binary instrumentation, such as the tools in pin [20] and in PEBIL [16], can be used for characterization and analysis with great level of detail by utilizing simulation, but cannot be used in a runtime system that controls data dynamically. With ADAMANT, we want to offer a range of capabilities that span from analysis to dynamic control of data movement.

3 ADAMANT

The main analysis tools in ADAMANT correspond to phases of the analysis process. The phases are: capturing the life cycle of data object, capturing data movement, infer data access patterns and search potential optimizations. To analyze an application, ADAMANT is preloaded when a program is first loaded into memory; this applies also to parallel programs in which each MPI process is attached to an instance of ADAMANT and produces a characterization file. Figure 1 shows the components of the application and ADAMANT. When loaded, ADAMANT has access to the application binary and from it, it reads the path to other dynamically linked libraries. ADAMANT itself is linked to characterization modules that collect relevant events during program execution. Throughout this process, ADAMANT maintains a database of data objects and associated events. Finally, the database of objects and related characterization information is stored in a file. This Section describes the components of ADAMANT and the three analysis phases.

The first step toward capturing and understanding data movement is to identify all the data objects of the application. ADAMANT captures the data objects used by the application and stores relevant information about them (e.g. number of references). Data objects are allocated in three ways: statically (e.g. in the data section of a program), dynamically (e.g. on the heap), and automatically (e.g. on the stack). Statically allocated data is identified immediately by reading the symbol tables in the binaries of the application and the libraries that are dynamically linked to it; the information collected forms the basis of the data object database that it maintains throughout execution. Dynamically allocated and automatically allocated data come into existence at runtime in a control-flow and input dependent manner. To capture dynamically allocated objects, ADAMANT defines wrapper functions for all standard memory allocation routines. Any time that an allocation takes place, the set of objects is

updated by inserting the new object, which is identified by its starting address. When an object is freed, the corresponding element in the set is marked as free. Finally, automatically allocated objects are represented as a whole by a *[stack]* object discovered by reading the memory mapping of the process. The current assumption is that automatic variables have a secondary role with respect global and dynamically allocated objects, and therefore the overhead of distinguishing between data objects corresponding to different variables and updating ADAMANT internal structures for every frame placed on the stack is not justified.

As data objects are identified, events that represent low-level data movement, such as transfer between DRAM and caches, are captured and associated to them. Events are captured by modules, as shown in Figure 1 (e.g. PEBIL), that either simulate such events or read hardware counters to detect their occurrence. Relevant events include memory references that trigger cache hits (or misses), and possibly access main memory. In addition, events carry information such as the instruction that triggered them and the address referenced. The latter is used to identify which object is accessed within the set of objects and to associate the events to it. While a program executes, the modules collect events and streams the events to ADAMANT, which associates events to data objects and collects per data object statistics. At the end, ADAMANT produces an execution profile for post-mortem analysis.

Finally, the focus is on creating a high level view of data movement useful for performance modeling, application and system tuning, and adaptive run-time strategies. To achieve such a view, data collected on data objects must be related to programming constructs and displayed in an intelligible way. To do so, we developed scripts to aggregate data into summary tables and bin objects by size or reference counts. This is still a primitive process and we continue to refine post-processing tools that parse the data collected by the instrumentation run and provide better per object summaries.

4 Case Studies

In this section we illustrate the use of ADAMANT in three case studies. In each case we compare three system configurations that employ non-volatile memory (NVM) to a base system, which includes no NVM (DRAM only, we refer to the this configuration as Base). In the first configuration, NVM is used as main memory and a DRAM pool of 16MB of memory per core is used as L4 cache (we refer to this configuration as 4LC); in the second configuration, the main memory is partitioned into a DRAM NUMA node and an NVM NUMA node, and memory objects can be allocated off either one (we refer to this configuration as hybrid main memory, HMM), and in the third, there is no DRAM and the main memory is a single NVM memory (we refer to this configuration as NMM). In all three configurations, it is assumed that the cost of accessing NVM is 3 times higher for reads and 5 times higher for writes than that of accessing DRAM. The profile collected using ADAMANT is used to model the average memory access time (AMAT), and determine which of the three configurations suffers the least slowdown with respect to a DRAM-only reference system [28].

4.1 Methodology

The studies are conducted by using the PEBIL binary instrumentation tool to simulate the cache hit rates of the different system configurations. PEBIL captures the address stream and passes it through a cache simulator configured to simulate an IvyBridge system (the base system) and the NVM configurations described. The corresponding cache and memory events from the simulator are streamed to ADAMANT, which associates these events to its data objects. This

process takes place at runtime (slowdown can vary and can be as high as 100x) and the address trace is not stored; rather, only the generated events are counted and associated to the objects. The cache configurations take into account that multiple processes share a cache by splitting the cache size evenly. The characteristics of the base system and the three NVM configurations are summarized in Table 1.

Name	L1	L2	L3	L4	DRAM main memory	NVM main memory
Base	32KB	256KB	2.5MB/core	0	4GB/core	0
4LC	32KB	256KB	2.5MB/core	16MB/core	0	4GB/core
HMM	32KB	256KB	2.5MB/core	0	16MB/core	4GB/core
NMM	32KB	256KB	2.5MB/core	0	0	4GB/core

Table 1: Memory system configurations. A 0 indicates that a memory is not present. 4GB/core of main memory is sufficient to hold the memory footprint of any the applications in the case studies.

ADAMANT's output is then used to model AMAT as shown in Equation 1. Equation 1 accounts for the latency of any load and store reference, as indicated by the count of the operations represented by variables *ld* and *st* which are multiplied by the corresponding latency variables *lat*; the sum is for each level of the hierarchy (subscript $_l$ in the sum) and to obtain the average, the sum is divided by the total number of memory references (ld_{tot} and st_{tot}). AMAT is then used to compare the relative performance of the different configurations.

$$AMAT = \frac{\sum_{l=1}^{mem} ld_l \times lat_l^{ld} + st_l \times lat_l^{st}}{ld_{tot} + st_{tot}} \quad (1)$$

4.2 NAS BT

The BT benchmark is a benchmark of the NAS parallel benchmark suite (NPB [4]). BT is part of the core CFD kernels of NPB and exhibits somewhat less parallelism and scalability than other CFD kernels; for this reason, using a smaller set of large-memory nodes is a viable option.

The benchmark is coded in Fortran and it allocates its variables statically. In this case, ADAMANT can easily find the data objects corresponding to the variables from the binary, including their names and size, as shown in Table 2.

Table 2 shows the objects with their size and the distribution of references (hit rates and other metrics are not shown for sake of simplicity). It should be noticed that most of the memory references (load and store operations) address the three largest objects but, unlike the stack and *work_1d*, which are accessed in the caches, *fields* is often reached in main memory. In addition, the latter is also so large that it accounts for most of the memory footprint, and in the HMM configuration, the only option is to allocate it in NVM.

The resulting AMAT has a slowdown of 2.13x for HMM relative to base, which is the same as if all objects were allocated in NVM (NMM). On the other hand, in the 4LC configuration, the slowdown is 2.02x, which is still quite high due to the poor locality in accessing *fields*. Depending on the optimization goals, employing a DRAM cache may lead to a small performance improvement; overall, the poor locality in accessing *fields* makes this application memory bound and therefore very sensitive to an increase in memory latency.

Objects	Size (B)	Footprint%	Ref%	Mld%	Mst%	Mem%
fields	1705261136	100.0	26.9	100.0	99.9	100.0
[stack]	135168	0.0	38.0	0.0	0.0	0.0
work_lhs	84824	0.0	27.1	0.0	0.1	0.0
work_1d	13696	0.0	1.7	1 0.0	0.0	0.0
constants	1280	0.0	6.4	0.0	0.0	0.0
partition	372	0.0	0.0	0.0	0.0	0.0
mpistuff	36	0.0	0.0	0.0	0.0	0.0
global	24	0.0	0.0	0.0	0.0	0.0

Table 2: Data objects of the BT benchmark sorted by size. The table also shows the percentage of memory references (Ref%) and the percentage of main memory accesses (Mem%) also divided by type (Mld% for loads, Mst% for stores).

4.3 Graph500-BFS

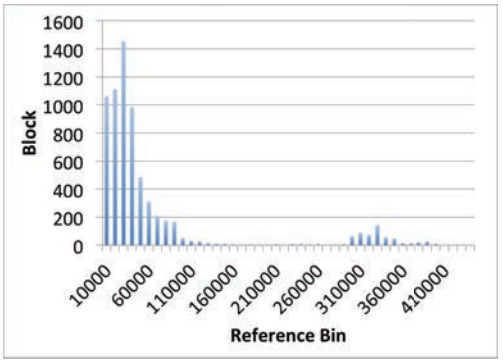
Graph500 is a benchmark designed to rank supercomputers for their ability to solve data-intensive problems [3]. Graph500 includes three kernels that are integer and memory intensive. One such kernel is the parallel Breadth-First-Search (BFS) kernel that we used in these experiments.

In the BFS kernel, a randomly generated graph is distributed across the nodes (also randomly), which then build a BFS tree in parallel. Table 3 shows the most relevant variables used in the BFS kernel. These variables are dynamically allocated and their names are discovered by recording the state of the stack at the time of allocation (this information is also associated to the data objects set maintained by ADAMANT) and by inspecting the code. In particular, we notice that there is a graph structure (*edgemem*), the stack, and then there are queues, variables, and buffers used for communication.

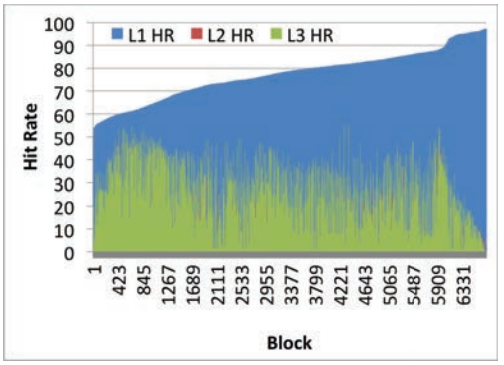
Objects	Size (B)	Footprint%	Ref%	Mld%	Mst%	Mem%
edgemem	134217728	88.6	0.4	88.0	0.0	73.3
has_edges	4194312	2.8	0.0	10.4	0.0	8.7
pred, g_oldq, g_newq	4194304	8.2	0.1	1.4	99.5	17.8
[stack]	135168	0.1	73.2	0.0	0.0	0.0
g_outgoing, g_visited	65536	0.0	0.7	0.1	0.5	0.2
g_recvbuf	4096	0.0	1.0	0.0	0.0	0.0
g_outgoing_counts, g_outgoing_reqs	128	0.0	1.3	0.0	0.0	0.0
g_outgoing_reqs_active	64	0.0	10.8	0.0	0.0	0.0
lgsize, rank, size	4	0.0	12.1	0.0	0.0	0.0

Table 3: Data objects of the BFS kernel sorted by size. The table shows the percentage of the memory footprint and the percentage of main memory accesses (Mem%) also divided by type (Mld% for loads, Mst% for stores).

edgemem is never modified by the BFS kernel, so while it occupies over 88% of the memory footprint, only loads access that object in memory. In addition, almost all of the remaining references that are not filtered by the caches involve four objects, of approximately 4MB each.



(a) Distribution of memory blocks in reference bins.



(b) Hit rates of all allocated blocks.

Figure 2: Blocks characterization: binning by reference and hit rates for all the blocks allocated.

We then compare the 4LC with HMM in which *has_edges, pred, g_oldq*, and *g_newq* are stored in DRAM. With 4LC, the slowdown is 1.09x relative to base, which is a slight improvement over the 1.10x for NMM. However, for HMM with the objects placement devised, the slowdown is 1.06x. In this case, a careful placement outperforms the LRU policy of the caches. Overall, while the memory footprint is large, there is enough locality in few objects to make this algorithm suitable for an NVM configuration with a DRAM cache, or even better a software managed memory pool.

4.4 Velvet

Velvet is a DNA assembler designed for short reads [30] that implements a *de novo* assembly algorithm. The structure of memory objects created in Velvet is very complex, because of the nature of the algorithm, and in fact it reflects the way the algorithm creates a large graph of small objects. The distribution of the thousands of objects created, divided into bins by their size and by the number of memory references, reveals that only few bins are important, in terms of objects, size, and memory references (not shown for sake of space). Nevertheless, it is very difficult to rely on the binning to devise a suitable placement policy. In particular, a careful analysis of the bins reveals that as much as 95.6% of the memory footprint is concentrated in a single size bin (512KB), and that the same size bin aggregates 96.3% of the references that reach main memory.

This is not a coincidence and in fact, by inspecting the code it was discovered that the application implements a custom memory allocation layer. The allocation layer allocates 128-page blocks of memory, that it then distributes to allocation requests within the code. As a result, neither the allocation point in the code (which is always the same) or the size is indicative of the use of the data objects. In these conditions it is hard to devise any placement policy.

Figure 2 shows the difference in references and locality between these blocks. Most of the 6739 blocks are referenced by less than 100000 instructions, and only few hundreds are referenced by more than 300000 instructions; also locality varies greatly: while the L2 hit rate is very low for all (not visible in the plot), there is great variability in L1 hit rates (from less than 50% to over 97%) and L3 hit rates.

For the 4LC configuration, the slowdown is 2.38x relative to base, whereas for NMM is 2.56x.

For HMM, we are limited to select 32 such blocks and place them in DRAM while placing the rest in NVM. The selection, which we based on the number of main memory operations in an attempt to reduce the number of NVM accesses, results in a 2.53x slowdown, which is a slight improvement over NMM. In this case, the characteristics of the allocator obfuscate any potential access pattern that could lead to a better placement, and the overall poor locality is reflected on the estimated slowdown.

5 Conclusions

We have described ADAMANT, a set of tools to capture, analyze, and manage data movement. As an application of ADAMANT, we demonstrated how it can be used to analyze programs and evaluate the impact of different memory configurations employing NVM memory as an alternative for DRAM in main memory.

The per-object information provided by ADAMANT is necessary for this kind of analysis in which information about objects cannot be aggregated and to preserve a per object view. In Graph500 case we are able to devise a data placement that outperforms a caching. In the other two cases, the understanding of data objects and access patterns made it possible to identify the sources of inefficiency (e.g. poor locality and sensitivity to memory latency); pointing out the objects involved provides a target for refactoring aiming at reducing the impact of slower memory technology.

To this end we continue to develop ADAMANT and extends its analysis capabilities. Beyond that, improving performance further would require a runtime system that dynamically manages objects in memory, migrating objects between NUMA nodes to direct as many references to the faster memory. Future work will also address this need by coordinating objects allocation, access, and placement dynamically. In addition, we are working on supporting other performance metrics and counters.

Acknowledgments

This work is supported in part by the NSF under award CCF 1451598, and by Intel Corporation.

References

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. Hpctoolkit: Tools for performance analysis of optimized parallel programs http://hpctoolkit.org. *Concurr. Comput. : Pract. Exper.*, 22(6):685–701, Apr. 2010.

[2] Advanced Micro Devices. *AMD64 Architecture Programmers Manual Volume 2: System Programming*. 2015.

[3] J. A. Ang, B. W. Barrett, K. B. Wheeler, and R. C. Murphy. Introducing the graph 500. In *Proceedings of Cray User's Group Meeting (CUG)*, May 2010.

[4] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The nas parallel benchmarks summary and preliminary results. In *Supercomputing, 1991. Supercomputing '91. Proceedings of the 1991 ACM/IEEE Conference on*, pages 158–165, Nov 1991.

[5] K. Beyls and E. H. D'Hollander. Discovery of locality-improving refactorings by reuse path analysis. In *Proceedings of the Second International Conference on High Performance Computing and Communications*, HPCC'06, pages 220–229, Berlin, Heidelberg, 2006. Springer-Verlag.

[6] L. Carrington, A. Snavely, X. Gao, and N. Wolter. A performance prediction framework for scientific applications. In *Proceedings of the 2003 International Conference on Computational Science: PartIII*, ICCS'03, pages 926–935, Berlin, Heidelberg, 2003. Springer-Verlag.

[7] P. Cicotti, L. Carrington, and A. Chien. Toward application-specific memory reconfiguration for energy efficiency. In *Proceedings of the 1st International Workshop on Energy Efficient Supercomputing*, page 2. ACM, 2013.

[8] S. Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013.

[9] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. *SIGPLAN Not.*, 38(5):245–257, May 2003.

[10] P. J. Drongowski. Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors, 2007.

[11] C. Fang, S. Carr, S. Önder, and Z. Wang. Reuse-distance-based miss-rate prediction on a per instruction basis. In *Proceedings of the 2004 Workshop on Memory System Performance*, MSP '04, pages 60–68, New York, NY, USA, 2004. ACM.

[12] A. Giménez, T. Gamblin, B. Rountree, A. Bhatele, I. Jusufi, P.-T. Bremer, and B. Hamann. Dissecting on-node memory access performance: A semantic approach. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 166–176, Piscataway, NJ, USA, 2014. IEEE Press.

[13] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. 2015.

[14] J. Jeffers and J. Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013.

[15] R. Lachaize, B. Lepers, and V. Quéma. Memprof: A memory profiler for numa multicore systems. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 5–5, Berkeley, CA, USA, 2012. USENIX Association.

[16] M. Laurenzano, M. Tikir, L. Carrington, and A. Snavely. Pebil: Efficient static binary instrumentation for linux. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pages 175–183, March 2010.

[17] X. Liu and J. Mellor-Crummey. Pinpointing data locality problems using data-centric analysis. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 171–180, Washington, DC, USA, 2011. IEEE Computer Society.

[18] X. Liu and J. Mellor-Crummey. A data-centric profiler for parallel programs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 28:1–28:12, New York, NY, USA, 2013. ACM.

[19] X. Liu and J. Mellor-Crummey. A tool to analyze the performance of multithreaded programs on numa architectures. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 259–272, New York, NY, USA, 2014. ACM.

[20] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6):190–200, June 2005.

[21] C. McCurdy and J. Vetter. Memphis: Finding and fixing numa-related performance problems on multi-core platforms. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 87–96. IEEE, 2010.

[22] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tool. *Computer*, 28(11):37–46, Nov. 1995.

[23] P. J. Mucci, S. Browne, C. Deane, and G. Ho. Papi: A portable interface to hardware performance counters. In *In Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[24] D. A. Patterson. Latency lags bandwidth. *Communication of ACM*, 47(10):71–75, 2004.

[25] K. O. Seager, A. Tiwari, M. A. Laurenzano, J. Peraza, P. Cicotti, and L. Carrington. Efficient hpc data motion via scratchpad memory. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, SCC '12, pages 801–805, Washington, DC, USA, 2012. IEEE Computer Society.

[26] S. S. Shende and A. D. Malony. The tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006.

[27] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '02, pages 409–, Washington, DC, USA, 2002. IEEE Computer Society.

[28] A. Suresh, P. Cicotti, and L. Carrington. Evaluation of emerging memory technologies for hpc, data intensive applications. In *Cluster Computing (CLUSTER), 2014 IEEE International Conference on*, pages 239–247. IEEE, 2014.

[29] W. A. Wulf and S. A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, Mar. 1995.

[30] D. R. Zerbino and E. Birney. Velvet: algorithms for de novo short read assembly using de bruijn graphs. *Genome research*, 18(5):821–829, 2008.